Breves Notas sobre
Análisis de Algoritmos

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Marzo, 2005

Índice general

Prefacio

Las *Breves Notas sobre Análisis de Algoritmos* introducen en forma simple y sencilla a algunos de los temas relevantes en el área de Análisis de Algoritmos. No tiene la intención de substituir a los diversos libros y publicaciones formales en el área, ni cubrir por completo los cursos relacionados, sino más bien, su objetivo es exponer brevemente y guiar al estudiante a través de los temas que por su relevancia se consideran esenciales para el conocimiento básico de esta área, desde una perspectiva del estudio de la Computación.

Los temas principales que se incluyen en estas notas son: Algoritmos, Corrección de Programas, Arboles de Cobertura Mínima, Multiplicación Rápida, el Problema de Repartición, Montones y Mezclas, Detectando Primos y Computación en Paralelo. Estos temas se exponen haciendo énfasis en los elementos que el estudiante (particularmente el estudiante de Computación) debe aprender en las asignaturas que se imparten como parte de la Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.

Jorge L. Ortega Arjona
Marzo, 2005

Capítulo 1

Algoritmos

Cocinando Programas

En la sintaxis exacta de un lenguaje de programación, un programa es la especificación de un proceso que se espera realice una computadora. La sintaxis es precisa y rigurosa. El más ligero error en la escritura del programa puede causar que el proceso sea erróneo o se detenga. La razón de esta situación parece paradójica en la superficie: es relativamente sencillo diseñar un programa que convierta la rígida sintaxis en un proceso; es mucho más difícil diseñar un programa que tolere errores o acepte un rango más amplio de descripciones de programas.

Un algoritmo puede especificar esencialmente el mismo proceso que un programa específico escrito en un lenguaje dado como Pascal o C. Aun así, el propósito de un algoritmo es comunicar el proceso no a las computadoras, sino a los seres humanos. Nuestras vidas (ya sea que trabajemos con computadoras o no) están llenas de algoritmos. Una receta de cocina, por ejemplo, es un algoritmo para preparar alimentos (suponiendo, por el momento, que se piensa en cocinar como una forma de proceso).

El propósito de este capítulo no es solo dar a conocer al lector la idea de un algoritmo, sino también introducirlo en alguna forma que puede utilizarse para presentar algoritmos. En la forma sencilla de una receta se presentan algunas convenciones comúnmente utilizadas por muchos estilos de descripción de algoritmos. La receta a continuación es simplemente para preparar enchiladas.

ENCHILADAS

1. *Precalentar el horno a 350 grados*
2. *En una cacerola*
   a) *Calentar dos cucharadas de aceite de oliva*
   b) *Freir media cebolla picada y un diente de ajo hasta dorar*
   c) *Añadir una cucharada de polvo de chile, una taza de puré de tomate y media taza de caldo de pollo*
   d) *Sasonar con sal y pimienta, y una cucharada de comino*
3. *Extender la salsa sobre tortillas*
4. *Llenar los centros con cantidades iguales de cebolla cruda picada y queso picado*
5. *Enrollar las tortillas*
6. *Colocarlas en un plato para horno*
7. *Vertir más salsa encima de las tortillas*
8. *Cubrir con trozos de queso*
9. *Calentar uniformemente en el horno por 15 minutos*

Comúnmente, el nombre del algoritmo se presenta al inicio, con letras mayúsculas: *ENCHILADAS*. Los pasos individuales o enunciados son usualmente (aunque no siempre) numerados. A este respecto, se hace notoria una característica peculiar de *ENCHILADAS*: algunos números se encuentran indentados, y usan numeraciones aparte. Por ejemplo, después del primer enunciado etiquetado con 2, hay cuatro enunciados etiquetados a, b, c y d, todos compartiendo la misma indentación. El propósito de la indentación es hacer visualmente más fácil reconocer que los siguientes cuatro pasos toman lugar en la cacerola que se especifica en el paso 2. Si se desea referirse a alguno de estos cuatro pasos, se usa la letra del paso precedida por el número del paso no indentado anterior. Por tanto, se espera que se añada cebolla picada y un diente de ajo en el paso 2.a. Tan pronto como las operaciones en la cacerola se han realizado al final del paso 2.d, las etiquetas de los pasos regresan a ser números no indentados. Este esquema de enumeración de enunciados resulta muy conveniente, ya que el propósito principal es hacer

posible su referencia. Es más fácil llamar la atención del lector en la línea 2.c en lugar de mencionar "el paso donde se añade polvo de chile, puré de tomate y caldo de pollo".

Las indentaciones usadas en la receta responden a una explicación más ámplia que las indentaciones hechas en la mayoría de los algoritmos o programas. Pero el enunciado 2.b es típico: hay una operación *continua* (freir) que se repite *hasta* que cierta condición se logra, es decir, cuando la cebolla y el diente de ajo se han "dorado". Tal tipo de enunciado se conoce como *ciclo*. Todos los enunciados que forman parte de un ciclo también se indentan.

Una característica importante que comparten *ENCHILADAS* con todos los algoritmos es la laxitud de su descripción. Revisando el algoritmo, es posible notar varias operaciones que no se mencionan explícitamente. ¿Qué tan grande debe ser la cacerola? ¿Cuándo se considera un color "dorado"? ¿Cuánta sal y pimienta debe usarse en el paso 2.d? Estas son cosas que podrían preocupar a un cocinero inexperto. Sin embargo, no preocuparían a cocineros expertos. Su juicio y sentido común llenan los posibles "hoyos" en la receta. Un cocinero experto, por ejemplo, sabe lo suficiente para sacar las enchiladas del horno al final del paso 9, aun cuando el algoritmo no incluye explícitamente tal instrucción.

Cualesquiera que sean los paralelismos entre cocina y computación, los algoritmos pertenecen más bien al segundo que al primero. Esto no quiere decir que los algoritmos sean incapaces de producir el equivalente computacional de una buena enchilada. Considérese, por ejemplo, el siguiente algoritmo gráfico, que se usa para producir una serie de dibujos en la pantalla de la computadora:

PATRON

1. **input** a, b
2. **input** $lado$
3. **for** $i \leftarrow 1$ **to** 100
    a) **for** $j \leftarrow 1$ **to** 100
        1) $x \leftarrow a + i * lado/100$
        2) $y \leftarrow b + j * lado/100$
        3) $c \leftarrow int(x^2 + y^2)$
        4) **if** $c\, es\, impar$ **then** $plot(i, j)$

Cuando se traduce este algoritmo a un programa para una computadora con alguna clase de dispositivo de salida gráfico (como es una simple pantalla) permite explorar una infinidad de patrones de decoración. Cada sección cuadrada del plano tiene un patrón único. Si el usuario da como entradas las coordenadas de la esquina inferior izquierda (a, b) de un cuadrado, seguida de la longitud de sus lados, el algoritmo dibuja una figura de decoración asociada a ese cuadrado. Sorprendentemente, si se escoge un cuadrado pequeño con las mismas coordenadas, emerge un nuevo patrón diferente, que no es una magnificación ni ninguna otra transformación del original.

Todo eso describe qué hace el algoritmo. Pero ¿cómo trabaja? Es notorio que hay dos ciclos **for** en el algoritmo. El ciclo más exterior en el paso 3 cuenta en forma continua de 1 a 100, usando la variable i para llevar el valor actual de la cuenta. Por cada uno de tales valores, el ciclo más interno cuenta de 1 a 100 a través de la variable j del paso 3.a. Más aun, para cada valor de j se repiten los cuatro pasos etiquetados de 3.a.1 a 3.a.4. Por cada repetición existen un nuevo par de valores i y j con los cuales se trabaja. Ahora bien, la variable x se calcula como una coordenada que es un centésimo del ancho de un cuadro, y se controla mediante la variable i. Similarmente, la variable y representa un centésimo de la altura de cada cuadrado, y se controla con la variable j. El punto resultante (x, y) se encuentra por lo tanto siempre dentro del cuadrado. Conforme j va de valor en valor, se puede pensar que el punto va ascendiendo en el cuadrado. Cuando llega a la parte superior $(j = 100)$, el ciclo interior se ha ejecutado por completo. En este punto, i cambia a su siguiente valor, y todo el ciclo interior comienza una vez más con $j = 1$.

El punto culmen del algoritmo consiste en calcular $x^2 + y^2$, una función circular, y luego tomar la parte entera (*int*) del resultado. Si el entero es par, un punto (digamos, blanco) se dibuja en la pantalla. Si el entero es impar, sin embargo, no se dibuja ningún punto para la coordenada (x, y), por lo que la pantalla permanece obscura (claro, si el fondo de la pantalla es originalmente obscuro).

El algoritmo $PATRON$ muestra varias prácticas comunes para expresar algoritmos:

- Los enunciados se indentan dentro del entorno de ciclos y enunciados condicionales del tipo **if...then...else**.
- Las asignaciones de un valor calculado a una variable se indican mediante flechas apuntando a la izquierda.

- El cálculo indicado en la parte derecha de una asignación puede ser cualquier fórmula aritmética.

Más aun, los enunciados del algoritmo *PATRON* son muy generales. Por ejemplo, se puede re-escribir el algoritmo como sigue:

PATRON

1. Entrada: *parámetros del cuadrado*
2. Por cada: *punto del cuadrado*
   a) Calcular su función *c*
   b) Si *c es par*, dibujarlo

Algoritmos en esta forma "comprimida" pueden servir para varios propósitos. Pueden representar un paso en el diseño de un programa más elaborado, en el cual el programador comienza con una descripción muy general y ámplia del proceso a realizar. Después, el programador re-escribe el algoritmo varias veces, refinándolo en cada paso. En algún punto, cuando el programa ha llegado a un nivel razonable de detalle, entonces puede traducirse en forma más o menos directa a un programa en cualquier lenguaje de programación, como Pascal o C.

Una segunda razón para escribir algoritmos "comprimidos" recae en el nivel de comunicación entre seres humanos. Por ejemplo, el escritor de un algoritmo normalmente comparte un cierto entendimiento con las demás personas acerca de la intención tras cada enunciado. Decir, por ejemplo, "Por cada *punto del cuadrado*" implica dos ciclos. Del mismo modo, la función *c* puede bien significar los pasos 3.a.1 a 3.a.d en el primer algoritmo *PATRON*.

El tipo de lenguaje algorítmico utiliza libremente construcciones disponibles en muchos lenguajes de programación. Por ejemplo, los siguientes tipos de ciclos son comúnmente utilizados:
for ... to
repeat ... until
while ...
for each ...

Además, existen enunciados de entrada (**input**) y de salida (**output**), y condicionales como **if ... then** e **if ... then ... else**. Ciertamente, se puede extender el lenguaje algorítmico para incluir cualquier construcción del lenguaje que parezca razonable. Las variables pueden ser números reales, valores lógicos (booleanos), enteros, o virtualmente cualquier entidad matemática con valor único. Se pueden tener arreglos de cualquier tipo o dimensión, listas, colas, pilas, etc. Aquéllos poco familiarizados con estas nociones pueden descubrir su significado en el contexto del algoritmo que los utilice.

La traducción de un algoritmo a un programa es generalmente directo, al menos cuando el algoritmo se encuentra razonablemente detallado. Por ejemplo, el algoritmo *PATRON* puede traducirse al siguiente programa en Pascal:

```
program PATRON (input, output);
var a, b, lado, x, y, c:real
var i, j:integer;
begin
    read(a,b);
    read(lado);
    graph mode;
    for i:=1 to 100
    begin
        for j:=1 to 100
        begin
            x:= a + i * lado/100;
            y:= b + j * lado/100;
            c:= trunc(x*x + y*y);
            if cmod2 = 0 then plot(i,j,1);
        end
    end
    textmode;
end
```

Muchos algoritmos incorporan preguntas sutiles e interesantes, como: ¿qué problemas pueden (o no) ser resueltos mediante algoritmos?, ¿cuándo es un algoritmo *correcto*?, ¿cuánto tarda en procesarse?

Capítulo 2

Corrección de Programas

Depuración Definitiva

El proceso de depurar (*debug*) un programa parece a veces eterno. Esto es especialmente cierto para algunos programas que no fueron analizados antes de ser escritos, o no fueron bien estructurados cuando se les escribió. Justamente cuando un programa parece estar ejecutándose correctamente, un nuevo conjunto de entradas da como resultado respuestas incorrectas. La correción de programas es una cuestión de importancia creciente en un mundo que cada vez depende más de la computadora. Para algunos programas de aplicación no es suficiente "suponer" de que se ejecutan correctamente; es necesario *estar seguro*.

La correción de programas es una disciplina que ha evolucionado a partir del trabajo pionero de C.A.R. Hoare en la Universidad de Oxford. Se basa en hacer ciertas afirmaciones (*assertions*) acerca de lo que programa debe haber cumplido en varias etapas de su operación. Las afirmaciones se prueban mediante razonamientos inductivos, a veces apoyados mediante análisis matemático. Si se prueba que un programa es correcto, se puede tener confianza en su operación, siempre y cuando la prueba sea correcta. Si la prueba falla, puede deberse a un error lógico (*logical bug*) en el programa, el cual la misma prueba ayuda a hacer notorio. En cualquier caso, el entendimiento del programa es muchas veces mejorado mediante intentar una prueba de su corrección.

En este capítulo se ilustra una prueba de corrección de un algoritmo euclideano en la forma de un programa. Este algoritmo encuentra el máximo común divisor (mcd) de dos enteros positivos. La figura 2.1 muestra dos enteros, 9 y 15, que se representan mediante barras horizontales divididas en unidades cuadradas.

El mcd de 15 y 9 es 3. En otras palabras, 3 es un divisor común de 15 y 9; es a la vez el máximo divisor. La barra etiquetada con 3, que se usa como si fuera una regla, muestra que es una medida entera de ambas, tanto de la barra 15 como de la

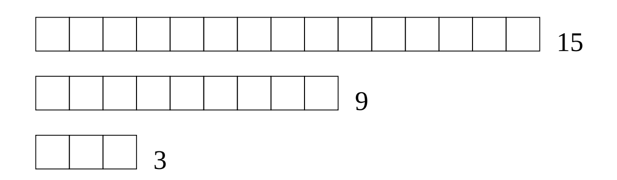


Figura 2.1: Dos enteros y su máximo común divisor

barra 9. Es también la regla más larga que tiene tal propiedad. Hace más de 2000 años, Euclides pudo haber usado una representación similar a ésta para descubrir el algoritmo que ahora lleva su nombre. Considérese, por ejemplo, otro conjunto de barras como el que se muestra en la figura 2.2. La barra 16 ajusta sólo una vez en la barra 22, y se tiene una barra 6 como restante. Ahora bien, comparando la barra 16 con la barra 6, es posible acomodar dos veces la barra 6 en la barra 16, y el restante es una barra de 4. El proceso se repite, ahora entre las barras 6 y 4. Sólo queda una barra 2, que al compararse de nuevo con la barra 4, no queda ningún restante. De esta forma, se obtiene que 2 es el mcd de 22 y 16.

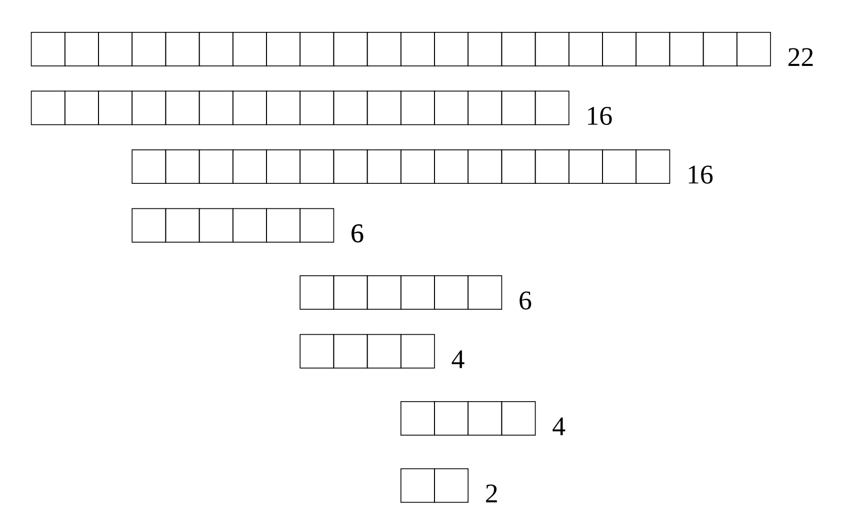


Figura 2.2: Otros dos enteros, 22 y 16

Este ejemplo sencillo ilustra el algoritmo euclideano. En cada paso, se toma el mayor número el cual se divide entre el menor número, y se calcula su residuo entero hasta que éste vale cero. En la forma de un algoritmo, se puede escribir como:

EUCLIDES

1. **input** M, N
2. **while** $M > 0$
   a) $L = N\, mod\, N$
   b) $N = M$
   c) $M = L$
3. **print** N

Una prueba de que este algoritmo produce el mcd de dos enteros positivos cualesquiera es más fácil de ilustrarse si se presenta el algoritmo en forma de un diagrama de flujo (figura 2.3).

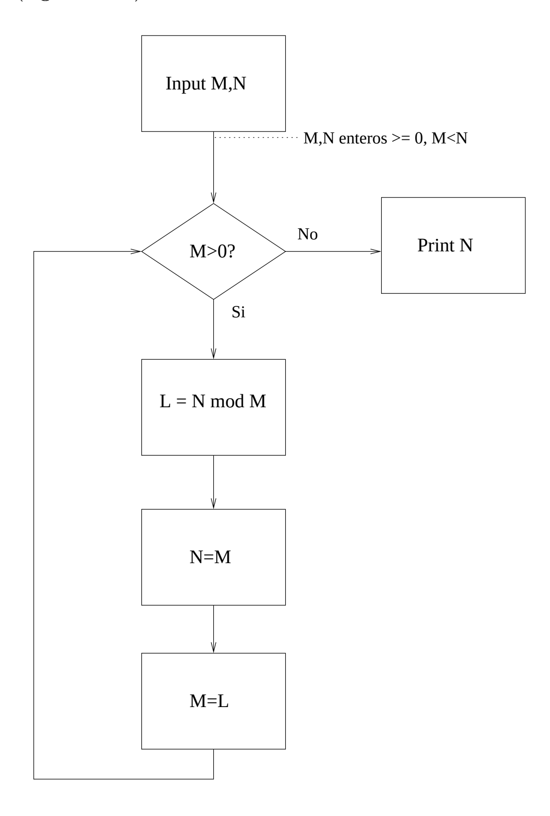


Figura 2.3: Diagrama de flujo para *EUCLIDES*

El diagrama de flujo ha sido etiquetado con una afirmación acerca de los valores de M y N que se introducen al programa: ambos valores son enteros positivos, y $M < N$. El algoritmo contiene un ciclo, y debido a que los valores de L, M y N

cambian con cada iteración, es posible distinguir los cambios en los valores conforme se realiza el proceso, mediante utilizar subíndices para las variables: denótese por L_i, M_i y N_i los valores de estas variables obtenidas en la i-ésima iteración, y supóngase que hay un total de k iteraciones. De este modo, es posible etiquetar el diagrama de flujo con otras afirmaciones, una acerca de lo que el algoritmo obtiene cuando sale del ciclo y otra acerca de los valores de las tres variables del algoritmo dentro del ciclo (figura 2.4).

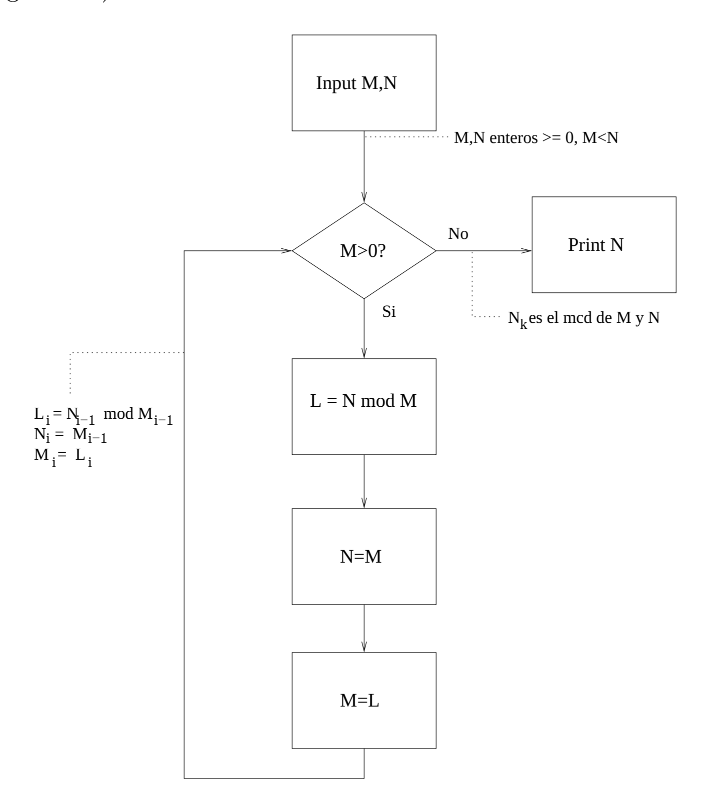


Figura 2.4: Diagrama de flujo para *EUCLIDES* con más afirmaciones

Claramente, se supone para empezar que $M > 0$, y que M_0 y N_0 representan los valores de entrada de M y N antes de comenzar con la primera iteración. Las etiquetas en el diagrama de flujo afirman no sólo que N_k es el último valor obtenido para N que es el mcd de M y N, sino que ocurren también ciertas relaciones que se mantienen entre los valores intermedios de L, M y N. Estas últimas afirmaciones son obviamente ciertas, ya que son re-enunciaciones de lo que significan las asignaciones dentro del contexto del ciclo. La primera afirmación es la que se busca probar.

En el caso cuando k (el número total de iteraciones) es 1, el algoritmo produce N_1, que resulta ser igual a M_0 de acuerdo con la etiqueta del ciclo. Pero M_1 debe

ser 0 para no entrar de nuevo al ciclo. A partir de las afirmaciones en el ciclo, se sigue que:

$$L_1 = M_1 = 0$$

y por lo tanto

$$N_0 \, mod \, M_0 = 0$$

En otras palabras, N es un múltiplo de M, y el mcd de M y N debe ser M mismo, que es el valor de N_1.

Si $k > 1$, entonces el ciclo debe tener al menos dos iteraciones sucesivas, y es posible hacer uso del siguiente resultado, descrito en la forma de un "lema". Aquí, se considera que $a \mid b$ significa "a divide a b sin residuo".

Lema: Dadas dos iteraciones sucesivas i e $i+1$ del ciclo, un entero positivo p satisface que $p \mid M_i \;\; y \;\; p \mid N_i$ si y solo si p satisface que $p \mid M_{i+1}$ y $p \mid N_{i+1}$.

Demostración: De acuerdo con las afirmaciones en el ciclo, $M_{i+1} = L_{i+1} = N_i \, mod \, M_i$. De esto se sigue que hay un entero positivo q tal que:

$$N_i = qM_i + M_{i+1}$$

Si $p \mid (M_i y N_i)$, entonces $p \mid M_{i+1}$ de acuerdo con esta igualdad. También $N_{i+1} = M_i$ para obtener $p \mid M_i$. Sin embargo, es claro que a partir de la igualdad que $p \mid N_i$, y el lema queda comprobado.

La principal implicación de este lema es que:

$$mcd(M_i, N_i) \mid mcd(M_{i+1}, N_{i+1})$$

y viceversa. Así, tenemos que:

$$mcd(M_i, N_i) = mcd(M_{i+1}, N_{i+1})$$

Ahora bien, de acuerdo a las afirmaciones en el ciclo, $N_k = M_{k-1} = N_{k-2} \, mod \, M_{k-2}$. Más aun, para terminar el ciclo, debe darse que $M_k = 0$, por lo que $L_k = 0$ y $N_{k-1} \, mod \, M_{k-1} = 0$. Esto significa que $M_{k-1} \mid N_{k-1}$. Pero $L_{k-1} = M_{k-1}$, y L_{k-1} es obviamente el mcd de M_{k-1} y N_{k-1}.

Sea L el mcd de M y N. Como el paso inicial de una demostración inductiva simple, es notorio que:

$$L = mcd(M_1, N_1)$$

Supóngase que para la i-ésima iteración, se tiene que:

$$L = mcd(M_i, N_i)$$

Entonces, por la conclusión obtenida del lema, se tiene que:

$$L = mcd(M_{i+1}, N_{i+1}) \qquad i < k.$$

El resultado se mantiene correcto hasta $i = k - 1$, pero en este caso ya se ha obtenido que $L_{k-1} = mcd(M_{k-1}, N_{k-1})$, y claramente $L = L_{k-1}$. Recuérdese que el valor de salida, N_k, es igual a L_{k-1}, por lo que se obtiene el resultado deseado.

Normalmente, al demostrar la corrección, es necesario probar también que el algoritmo termina su ejecución para todos las entradas de interés. En el caso del algoritmo *EUCLIDES*, se podría probar la terminación mediante notar, en efecto, que M decrece con cada iteración, y dado que M es inicialmente un entero positivo, eventualmente debe llegar al valor de cero. Es aquí donde se toma en cuenta la no-negatividad de M y N.

Para los algoritmos con más de un ciclo, y especialmente cuando los ciclos son anidados, es necesaria una estrategia de comprobación más completa. En general, además de las afirmaciones sobre los valores de entrada y de salida, al menos una afirmación debe aparecer por cada ciclo del algoritmo. Es entonces necesario comprobar para cada ruta entre dos afirmaciones adyacentes A y A' que si A es verdadera cuando el algoritmo alcance ese punto, entonces A' será verdadera cuando la ejecución llegue ahi. Este requerimiento se cumple de manera obvia en el caso de la comprobación del algoritmo *EUCLIDES*, que contiene tan solo un ciclo.

Capítulo 3

Arboles de Cobertura Mínima

Un Algoritmo Veloz

El área conocida como Teoría de Grafos es una rama de las matemáticas que tiene una relación especial con la computación, tanto en aspectos teóricos como prácticos: el lenguaje, las técnicas y los teoremas que conforman la Teoría de Grafos, así como el hecho de que la Teoría de Grafos es en sí misma es una fuente rica de problemas que representan un reto para resolverlas utilizando una computadora. Ciertamente, no muchos de los problemas de esta teoría parecen tener algún algoritmo que los resuelva en *tiempo polinomial*. De hecho, muchos de los primeros problemas en demostrarse ser NP-completos fueron problemas de la Teoría de Grafos.

De entre los problemas bien conocidos y ya resueltos, se encuentra la búsqueda de un árbol de cobertura mínima para una grafo. Específicamente, dado una grafo G, con aristas de varias longitudes, el problema es encontrar un árbol T en G tal que:

- T "cubra" G, es decir, todos los vértices de G se encuentren en T.
- T tiene la longitud mínima total, sujeto a la condición anterior.

El árbol (que se muestra en líneas más gruesas) de la figura 3.1 cubre al grafo que se muestra, pero no se trata de un árbol de cobertura mínima. Por ejemplo, si uno de las aristas incidentes con un vértice marcado con un círculo se remueve del árbol, y la arista que une los dos vértices marcados con un círculo se le añade, entonces el árbol resultante aún cubre al grafo, y tiene una longitud menor al árbol original. Las preguntas que surgen son entonces: ¿cómo se encuentra el árbol de cobertura mínima de un grafo?, ¿existe más de uno?

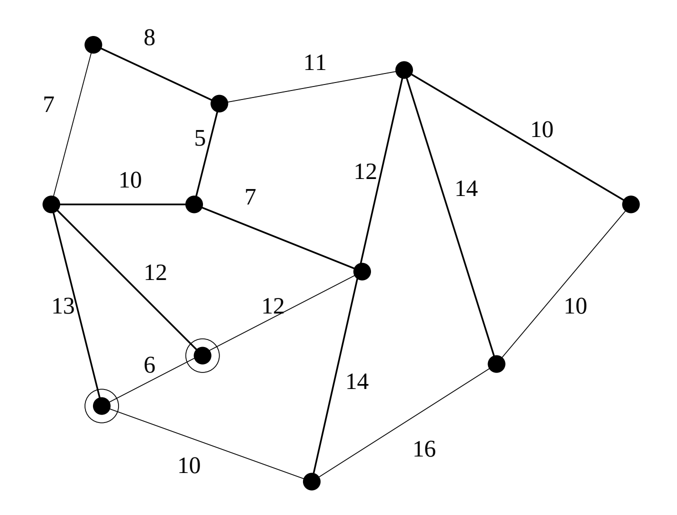


Figura 3.1: Un grafo y un árbol de cobertura

El algoritmo conocido más eficiente para hallar el árbol de cobertura mínima resulta ser un algoritmo "voraz". Tal tipo de algoritmos resuelven problemas de optimización mediante optimizar cada paso, del mismo modo que una persona voraz cuando se le presenta un plato de galletas: cada vez que se le permite a la persona seleccionar una galleta, ésta siempre selecciona la más grande.

El algoritmo para encontrar el árbol de cobertura mínima que se presenta a continuación fue desarrollado por primera vez en 1956 por George Krusal, un matemático estadounidense. Procede mediante "hacer crecer" un árbol de cobertura una arista cada vez. Debido a que el árbol debe tener una longitud mínima, el algoritmo siempre selecciona la arista disponible más corta para añadir al árbol. En este sentido, el algoritmo es "voraz".

El algoritmo, llamado *MINSPAN*, usa un lista L de aristas que unen vértices del árbol bajo construcción con aquellas aristas que no han sido aún cubiertas. El árbol en sí mismo se denota por T, y por cada vértice v, E_v representa el conjunto de todas las aristas incidentes en v.

MINSPAN

1. Seleccionar un vértice arbitrario u en el grafo.
2. $T \leftarrow \{u\}$
3. $L \leftarrow E_u$
4. $L \leftarrow \text{ordena}(L)$
5. **while** T no cubra G

a) Selecciona la primer arista $\{v, t\}$ en L

b) $T \leftarrow T \bigcup \{v\} \bigcup \{v, t\}$

c) $L' \leftarrow E_v - L$

d) $L \leftarrow L - E_v \bigcap L$

e) $L' \leftarrow \text{ordena}(L')$

f) $L \leftarrow \text{mezcla}(L, L')$

Inicialmente, T consiste de un solo vértice u, seleccionado arbitrariamente. La lista L en principio contiene todas las aristas coincidentes en u. Estas aristas se ordenan de acuerdo a su longitud. El algoritmo procede entonces iterativamente a añadir la arista disponible más corta al árbol, una a la vez. La arista más corta $\{v, t\}$ es fácil de calcular, ya que siempre es el primer miembro de L; une un vértice t en T al vértice v que no se encuentra en T. En los pasos 5.b, 5.c y 5.d, el vértice v y la arista $\{v, t\}$ se añaden a T, se genera la lista L' de nuevas aristas para añadirse a L, y entonces L misma se reduce a todas las aristas que unen v a algún otro vértice en T. Se crea una nueva lista ordenada L en el paso 5.f, cuando la lista ordenada anterior se mezcla con la lista ordenada L' de nuevas aristas.

Es interesante examinar el árbol de cobertura generado por *MINSPAN* en casos específicos. Por ejemplo, *MINSPAN* produce el árbol que se muestra en la figura 3.2, para el grafo de la figura 3.1. Se circula el vértice inicial u.

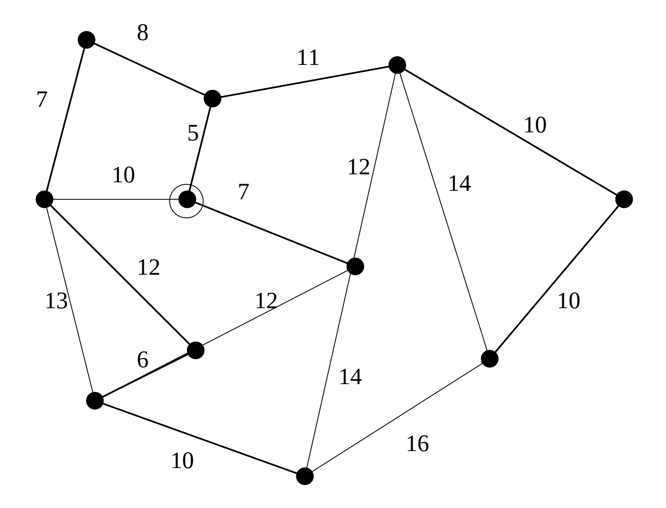


Figura 3.2: Un árbol de cobertura mínima hallado por *MINSPAN*

En el análisis de algoritmos, generalmente se involucran dos pasos. Primero, debe probarse que el algoritmo es correcto. Segundo, la complejidad del algoritmo

debe establecerse tan precisamente como sea posible, dentro de un orden de magnitud. Cuando se comprueba la corrección de un algoritmo, el argumento puede ser relativamente informal. Al probar la corrección de programas, sin embargo, la precisión de un lenguaje de programación específico produce pruebas más rigurosas en lo posible. El mismo tipo de mención puede hacerse acerca de establecer la complejidad en tiempo de un algoritmo. Cuando un algoritmo se describe en mayor detalle, puede ser analizado en mayor profundidad, algunas de las veces resultando en una cantidad de complejidad diferente en orden de magnitud.

¿Cómo se sabe que *MINSPAN* realmente obtiene un árbol de cobertura mínima para un grafo G? Se puede probar por inducción que *MINSPAN* trabaja correctamente. Primero, si G tiene un solo vértice, entonces *MINSPAN* termina con ese único vértice como el árbol de cobertura mínima. Segundo, se supone que *MINSPAN* siempre encuentra un árbol de cobertura mínima para grafos con $n-1$ vértices, suponiendo que G tiene n vértices y examinando la operación de *MINSPAN* justo antes de que el último vértice w de G se añada a T. Se define a H como el subgrafo de G obtenido al borrar del grafo G el vértice w junto con todas sus aristas incidentes. Se hace operar *MINSPAN* sobre el subgrafo H, precisamente como se ha hecho con G, ya que la remoción de w no acorta ninguna de las aristas que quedan. Se sigue entonces de este hecho y de la hipótesis de inducción que T es un árbol mínimo para H. En su última iteración sobre G, *MINSPAN* añade la arista final que une w con T, y esta arista es la más corta de todas las aristas disponibles. Ahora bien, si el árbol resultante T' no es mínimo en G, entonces hay un árbol de cobertura mucho menor T'' para G. Sea $\{w, v\}$ la arista de T'' que contiene a w, y véase que $T'' - w$ debe ser un árbol de cobertura para H que debe ser aún más corto que T. Esta contradicción establece el resultado.

Habiendo producido un argumento inductivo razonablemente congruente de que *MINSPAN* siempre encuentra un árbol de cobertura mínima para un grafo dado, se establece ahora una cantidad en orden de magnitud de la complejidad en tiempo de *MINSPAN*.

Hasta ahora, no se ha sido muy preciso en cuanto a qué tipo de estructuras de datos usa *MINSPAN*. Resulta ser más eficiente almacenar tanto G como a T en forma de listas de aristas de acuerdo con el siguiente formato:

$$v_i : v_{i1}, c_{i1}; v_{i2}, c_{i2}; \ldots$$

En otras palabras, usando un arreglo o una lista ligada, se almacena el vértice v_i junto con los vértices v_{ij}, en donde $\{v_i, v_{ij}\}$ es una arista de G y c_{ij} es la longitud de tal arista. El análisis de la complejidad en tiempo procede ahora paso por paso:

- El paso 1, seleccionar un vértice arbitrario de G, tiene un costo de 1 unidad de tiempo.
- Los pasos 2 y 3, añadir u a la lista (inicialmente vacía) que define a T, tiene un costo de 1 unidad de tiempo, y leer E_u de la lista tiene un costo d_u, donde

d_u es el número de aristas incidentes con u.

- El paso 4, ordenar la lista L mediante algún método razonablemente eficiente como *mergesort* tiene un costo de $d_v \, log \, d_v$ unidades de tiempo.
- En el ciclo del paso 5, comprobar si T cubre a G tiene un costo de 1 unidad de tiempo.
- En el paso 5.a, ya que L se ordena en forma ascendente, la primer arista de L es $\{v, t\}$, y tiene un costo de 1 unidad de tiempo para obtenerse.
- En el paso 5.b, la lista que define T no tiene un orden especial, y los nuevos vértice y arista pueden añadirse en 1 unidad de tiempo.
- En el paso 5.c, el algoritmo *MINSPAN* debe examinar cada arista en E_v, y decidir si está el L. Ya que las aristas en L están ordenadas, tiene un costo de $log \mid L \mid$ unidades de tiempo para cada decisión. Esto arroja un total de $d_v \, log \mid L \mid$ unidades de tiempo.
- En el paso 5.d, para eliminar las aristas $E_v \bigcap L$ de L se requiere de nuevo llevar a cabo d_v búsquedas en L para hallar los miembros de E_v que se encuentran en L. Cada búsqueda tiene un costo de $log \mid L \mid$ unidades, y cada eliminación cuenta 1 unidad de tiempo. Por lo tanto, este paso tiene un costo de $d_v \, log \mid L \mid +1$ unidades de tiempo.
- En el paso 5.e, el ordenamiento de L' tiene un costo no mayor que $d_v \, log \, d_v$ unidades de tiempo.
- En el paso 5.f, la cantidad de tiempo requerido para mezclar las listas L' y L involucra al menos d_v búsquedas e inserciones en L, dando un total de $d_v \, log \mid L \mid +1$ unidades de tiempo.

Esto completa un análisis detallado del algoritmo. Es necesario ahora sumar las cantidades en forma determinada. Para esto, se etiqueta a los vértices de G en el orden en que aparecen en T, es decir, $v_1, v_2, v_3, ...$ Coherentemente, sus costos se indican $d_1, d_2, d_3, ...$, y en la i-ésima iteración de *MINSPAN*, L no puede ser mayor a $d_1 + d_2 + d_3 + ... + d_{i-1}$, en tanto que $\mid E_v \mid = d_i$. Esto produce las siguientes expresiones:

- Para los pasos 1 a 4:
  $2 + d_1 + d_1 \, log d_1$
- Para los pasos 5 y de 5.a a 5.d:
  $4 + 2d_i \, log \, (d_1 + d_2 + d_3 + ... + d_{i-1}) + d_i$
- Para los pasos 5.e y 5.f:
  $d_i \, log \, d_i + d_i \, log(d_1 + d_2 + d_3 + ... + d_{i-1}) + d_i$

La expresión final se obtiene de sumar la primera expresión con la suma iterada de las expresiones que quedan:

$$2 + d_1(log\, d_1 + 1) + \sum_{i=2}^{n} 4 + 3d_i\, log\,(d_1 + d_2 + d_3 + ... + d_{i-1}) + 2d_i + d_i\, log\, d_i$$

Simplificando un poco, se puede probar que esta expresión se limita superiormente por una expresión más simple como:

$$m\, log\, 2m + 4n$$

donde m es el número de aristas en G y n es el número de vértices. Suponiendo que G es un grafo conexo, se tiene que $m \geq n$, por consiguiente, el límite superior mínimo es $O(m\, log\, m)$. Esto da la complejidad en tiempo de *MINSPAN* en función del número de aristas en G.

Con esto, se termina el análisis de *MINSPAN*. Además de ser un algoritmo correcto y bastante eficiente, muestra la simplicidad básica y elegancia de algunos de los problemas mejor resueltos en Teoría de Grafos. La idea esencial es simplemente hacer "crecer" un árbol de cobertura mediante añadir la arista más corta a la vez.

Es interesante que virtualmente el mismo algoritmo puede ser utilizado para encontrar el árbol de cobertura máxima. Se necesita sólo alterar la instrucción 5.a, diciendo "Selecciona la última arista $\{v, t\}$ en L". Sin embargo, en el caso de otro problema cercanamente relacionado, el de encontrar la ruta más corta entre dos vértices, la situación es totalmente diferente. No hay forma, aparentemente, de alterar el algoritmo de la ruta más corta para encontrar la ruta más larga.

Hay otro problema con árboles mínimos cercanamente relacionado al que se expone aquí. Supóngase un grafo dado G, y un subconjunto específico S de los vértices de G. ¿Cuál es el sub-árbol de longitud mínima en G que cubre todos los vértices en S? Tal árbol podría ciertamente querer utilizar algunos de los vértices en G que no se encuentran en S, pero no se require, en general, cubrir todos los vértices en G. Ese árbol mínimo en G se conoce como un *árbol de Steiner*, y el problema de encontrarlo eficientemente resulta mucho más difícil de resolver que el problema de los árboles de cobertura mínima.

Capítulo 4

Multiplicación Rápida

Divide y Conquista

Sería interesante estimar el número de multiplicaciones que se realizan diariamente por computadoras alrededor del mundo. Tal número se encuentra probablemente entre 10^{15} y 10^{20} en la actualidad, considerando la población mundial de computadoras. En la mayoría de estas máquinas, la multiplicación absorbe tan solo unos cuantos microsegundos. Los circuitos electrónicos que realizan la multiplicación se han optimizado para producir el producto de, digamos números de 32 bits, en el menor tiempo posible. Sin embargo, la disponibilidad de soluciones rápidas en el hardware al problema de multiplicar números de 32 bits tiende a obscurecer una pregunta muy general que podría tener más que una importancia meramente teórica: ¿Qué tan rápido se pueden multiplicar dos números de n bits? En lugar de intentar utilizar un esquema de hardware generalizado para la multiplicación rápida, se supone que todas las operaciones se realizan a nivel de bit, y entonces, meramente se intenta determinar el menor número de operaciones de bit necesarias para formar los productos.

El contexto a nivel de bit de este problema puede ilustrarse mediante considerar por el momento la adición binaria:

$$\begin{array}{r} 1\,0\,0\,1 \\ +\;\;1\,1\,0\,1 \\ \hline 1\,0\,1\,1\,0 \end{array} \qquad \begin{array}{ccccc} & 1 & 0 & 0 & 1 \\ & + & + & + & + \\ & 1 & 1 & 0 & 1 \\ & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 \leftarrow & 0 & 1 & 1 \leftarrow & 0 \end{array}$$

Al sumar los dos números 1001 y 1101, las operaciones sobre los bits individuales se representan en las columnas. Comenzando por el extremo a la derecha, se suman 1 más 1 para obtener 0, y propagar un acarreo a la siguiente columna. Nótese que cada

una de estas operaciones se consideran exclusivamente sobre bits individuales. Es claro que, sin importar qué tan largos sean los números a sumar, estas operaciones pueden considerarse como fundamentales, y absorben, digamos, 1 unidad de tiempo. Igualmente claro es que para sumar dos números de n dígitos binarios se requieren n de tales operaciones.

En el caso de la multiplicación, sin embargo, la situación es muy diferente. Primero, adaptando las reglas ordinarias para la multiplicación que se aprenden en la escuela primaria, no es difícil notar que dos números de n bits pueden multiplicarse en $n^2 + 2n - 1$ unidades de tiempo.

$$\begin{array}{rrrrrrrr}
 & & & & 1 & 0 & 0 & 1 \\
\text{x} & & & & 1 & 1 & 0 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 1 \\
 & & & 0 & 0 & 0 & 0 & \\
 & & 1 & 0 & 0 & 1 & & \\
 & 1 & 0 & 0 & 1 & & & \\
\hline
 & 1 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}$$

Se requieren n^2 pasos para formar los n productos intermedios, y otros $2n - 1$ pasos para sumarlos. En la práctica, se ignoran los $2n - 1$ pasos, concentrándose en sólo en los n^2 pasos, mediante expresar que esta clase de multiplicación requiere "en order de" n^2 unidades de tiempo, o simplemente:

$$O(n^2)\, pasos$$

Sería lo más inmediato intentar reducir la potencia de n de esta expresión. ¿Podría, por ejemplo, reducirse a $O(n^1)$?

Como un intento inicial para acelerar la multiplicación a nivel bit, se toma una aproximación conocida como "divide y conquista": supónganse dos números de n bits x y y, que se multiplicarán, y que cada uno de ellos se divide en dos partes de igual longitud, considerando lo siguiente:

$$x = a \times 2^{n/2} + b$$
$$y = c \times 2^{n/2} + d$$

El proceso de partición se realiza mediante revisar la mitad de cada número y meramente dividirlo en dos partes (en caso de no encontrarse la "mitad", se puede simplemente añadir un 0 como bit más significativo). Por ejemplo:

$$1001 = 10 \times 2^2 + 01$$
$$1101 = 11 \times 2^2 + 01$$

Habiendo dividido x y y en dos partes cada uno, es posible escribir la multiplicación de tales números como un producto de cuatro partes:

$$x \times y = (a \times c) \times 2^n + (b \times c) \times 2^{n/2} + (a \times d) \times 2^{n/2} + (b \times d)$$

A primera vista, parece que se requiere resolver cuatro multiplicaciones de números con $n/2$ bits, y que serán necesarias $4(n/2)^2 = n^2$ multiplicaciones. Esto no parece ser un buen comienzo.

Sin embargo, si $(a \times c)$ y $(b \times d)$ estuvieran calculadas de antemano, entonces no sería necesario realizar dos de las multiplicaciones para obtener los productos $(b \times c)$ y $(a \times d)$. De hecho:

$$(b \times c) + (a \times d) = (a + b) \times (c + d) - (a \times c) - (b \times d)$$

¿Cuántas operaciones a nivel bit se requieren para realizar los procesos implícitos en estas observaciones? Para obtener a y b a partir de x se requieren $n/2$ operaciones a nivel bit en cada caso, y se supone que, por el momento al menos, los productos $(a \times c)$ y $(b \times d)$ se obtienen por el "método de la escuela primaria", así que requieren cada uno $(n/2)^2 + 2(n/2) - 1$ operaciones a nivel bit. Las adiciones $(a+b)$ y $(c+d)$ requieren cada una $n/2$ operaciones, ya que cada una se realiza sobre números de $n/2$ bits; el producto de estas dos cantidades requiere $(n/2+1)^2 + n + 1$ de tales pasos. En resumen, la tabla siguiente muestra toda esta información:

Operación	**Número de pasos**	**Número de bits del resultado**
a y b	$n/2$ cada uno	$n/2$ cada uno
$a \times c$ y $b \times d$	$(n/2)^2 + 2(n/2) - 1$ cada uno	n cada uno
$a + b$ y $c + d$	$n/2$ cada uno	$n/2 + 1$ cada uno
$(a + b) \times (c + d)$	$(n/2 + 1)^2 + 2(n/2 + 1) - 1$	$n + 2$
$(a + b) \times (c + d)$ $-a \times c - b \times d$	$2n$	$n + 2$

En el último renglón de la tabla, el producto $(a + b) \times (c + d)$ se encuentra disponible como un número de $n+2$ bits, mientras que $a \times c$ y $b \times d$ están disponibles en su forma de n bits. De tal forma, sumar los términos (negativos) segundo y tercero al primer término requiere de un total de $2n$ operaciones, sin contar la propagación de acarreos. Sumando el número de pasos total en la forma de un resultado final, se tienen $3(n/2)^2 + 15(n/2)$ operaciones a nivel bit.

Esta cantidad llega a ser cercana a $3n^2/4$, pero, y a menos que se tenga la idea creativa de usar la misma técnica de nuevo para el cómputo de $a \times c$, $b \times d$ y $(a + b) \times (c + d)$, únicamente se lograría una mejora en términos de un factor constante. Ciertamente, en éste como en muchos otros problemas, es necesario dividir una y otra vez antes de llegar a "conquistarlo".

Ahora bien, sea $T(n)$ que denota el número de operaciones de bit que se pueden lograr de esta forma en la multiplicación de dos números de n bits. Mediante simplemente insertar $T(n/2)$ y $T(n/2+1)$ en los sitios adecuados de la tabla anterior, se obtienen las siguientes expresiones recursivas:

$$\begin{aligned} T(n) &= 2T(n/2) + T(n/2+1) + 8(n/2) \\ &= 3T(n/2) + cn \end{aligned}$$

donde c es una constante apropiadamente escogida.

Es entonces más sencillo mostrar por el método de inducción que:

$$T(n) \leq 3cn^{log3} - 2cn$$

Esta cota superior de la velocidad con la que dos números de n bits se multiplican puede re-escribirse como $O(n^{1,59})$. Este valor aún no llega a $O(n^1)$, pero representa una verdadera mejora sobre $O(n^2)$.

La técnica de multiplicación rápida que se ha descrito hasta ahora fue originalmente descrita por A. Karatsuba y Y. Ofman en 1962. Fue el mejor resultado de su tipo hasta que fue mejorada en 1971 por una técnica descubierta por A. Schönhage y V. Strassen, quienes desarrollaron un algoritmo de divide y conquista que requiere tan solo $O(n\, log\, nlog\, log\, n)$ operaciones a nivel bit, representando una mejora muy cercana al objetivo (que tal vez no sea lograble) de $O(n^1)$.

Considérese ahora el problema de multiplicar dos matrices de $n \times n$. En el análisis siguiente ya no se examinan operaciones a nivel de bit, ya que n no representa el número de bits, sino mas bien, el tamaño de la matriz. Al adoptar tal suposición, es necesario suponer también una computadora que realiza multiplicaciones tan rápido como realiza adiciones. De cualquier modo, esto no es del todo irreal, ya que *(a)* la mayoría de las computadoras tienen un circuito paralelo de alta velocidad para la multiplicación; *(b)* los valores de la matriz se suponen tales que quepan en una palabra de cualquier computadora que se esté utilizando; y *(c)* no importa que factor constante separe el tiempo de una multiplicación del tiempo de una adición, se obtendrá el mismo valor en orden-de-magnitud como una función de n.

El método básico de multiplicar dos matrices X y Y es utlizar directamente la definición: el ij-ésimo elemento del producto se obtiene por:

$$\sum_{k=1}^{n} x_{ik} y_{kj}$$

donde claramente se requieren $O(n^3)$ operaciones, entre adiciones y multiplicaciones, y de las cuales se requieren tan solo $O(n)$ operaciones para generar cada uno de los n^2 productos.

Al mismo tiempo, ya que el producto $Z = X \times Y$ tiene n^2 productos, es de esperar que no se pueda realizar la operación con un número menor de operaciones que $O(n^2)$. Para lo que sigue, y por facilidad, se considera que n es potencia de 2. Si no lo fuera, de cualquier modo es posible "rellenar" las matrices con ceros, de tal modo que lo fuera. A pesar de esto, el producto Z de ambas matrices permanece inalterable dentro del producto de las matrices aumentadas.

Como un primer paso, considérese que X, Y y Z se parten en matrices de $n/2 \times n/2$ de la siguiente forma:

$$\begin{pmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{pmatrix} = \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \times \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix}$$

Una manipulación algebraica permite formar las matrices intermedias:

$$\begin{aligned}
W_1 &= (X_{11} + X_{22}) \times (Y_{11} + Y_{22}) \\
W_2 &= (X_{21} + X_{22}) \times (Y_{11}) \\
W_3 &= (X_{11}) \times (Y_{12} + Y_{22}) \\
W_4 &= (X_{22}) \times (Y_{11} + Y_{21}) \\
W_5 &= (X_{11} + X_{12}) \times (Y_{22}) \\
W_6 &= (X_{21} - X_{11}) \times (Y_{11} + Y_{12}) \\
W_7 &= (X_{12} - X_{22}) \times (Y_{21} + Y_{22})
\end{aligned}$$

de modo que las cuatro submatrices de Z pueden obtenerse mediante las siguientes adiciones:

$$\begin{aligned}
Z_{11} &= W_1 + W_4 - W_5 + W_7 \\
Z_{12} &= W_3 + W_5 \\
Z_{21} &= W_2 + W_4 \\
Z_{22} &= W_1 + W_3 - W_2 + W_6
\end{aligned}$$

De haber calculado las submatrices Z_{ij} directamente como productos matriciales de X_{ij} y Y_{ij}, se requeriría de ocho multiplicaciones de matrices. El método anterior requiere únicamente de siete. Partiendo de esto, sea $T(n)$ el número total de operaciones requeridas de una estrategia divide-y-conquista. La expresión resultante de $T(n)$ se vuelve:

$$T(n) = 7T(n/2) + 18(n/2)^2$$

Utilizando inducción y el hecho de que $T(1) = 1$, se puede resolver que esta recurrencia tiene el siguiente número de operaciones:

$$\begin{aligned} T(n) &= 7^{log\, n} + 18n^2 \sum_{k=0}^{\lfloor log\, n \rfloor} (7/4)^k \\ &\doteq 19n^{log\, 7} \\ &= O(n^{2,81}) \end{aligned}$$

En el caso de la multiplicación de matrices, es notorio que no ha habido una mejora dramática como ocurre en el caso de la multiplicación de enteros. La técnica que obtiene $O(n^{2,81})$ descrita anteriormente fue desarrollada por V. Strassen en 1969. Dos mejoras le han seguido, ambas durante 1979. En la primera, A. Schönhage obtuvo un método que requiere $O(n^{2,73})$, mientras que la segunda, por V. Pan, propone un método con $O(n^{2,61})$. ¿Será este último orden-de-magnitud el mejor posible?

Capítulo 5

El Problema de Repartición

Un Algoritmo Pseudoveloz

Ocasionalmente, alguien que trabaja con un problema NP-completo imagina haber descubierto un algoritmo exacto y que se ejecuta en tiempo polinomial para tal problema. Hasta ahora, comúnmente tal persona se encuentra errada. Sin embargo, esto hace notar no sólo la necesidad de un análisis cuidadoso de nuevos algoritmos, sino también el hecho de que la propiedad de un algoritmo de ser NP-completo puede ser realmente sutil.

Tal es el caso del *problema de la repartición*: Dados n enteros $x_1, x_2, \ldots, x_n$, se requiere encontrar una repartición de estos enteros en dos conjuntos I y J, tales que:

$$\sum_{i \in I} x_i = \sum_{j \in J} x_j$$

En otras palabras, ambos subconjuntos de enteros deben sumar la misma cantidad. Una analogía simple para este problema puede hacerse mediante bloques de varias alturas (figura 5.1). ¿Es posible construir dos columnas de bloques con la misma altura?

El problema de la repartición puede simplificarse en algo antes de intentar darle una solución algorítmica. En forma razonable, se puede sumar las alturas de todos los bloques y tratar de construir una sola columna cuya altura fuera de la mitad de la suma. Sin embargo, aún al intentar esta simple tarea, es necesario probar sistemáticamente con todos y cada uno de los subconjuntos posibles del conjunto de bloques. Tal procedimiento consumiría una gran cantidad de tiempo. Ciertamente, dado un conjunto de bloques, no hay garantía alguna de que la solución al problema planteado exista, como podría ser el caso, por ejemplo, de que la suma de las alturas fuera un número impar.

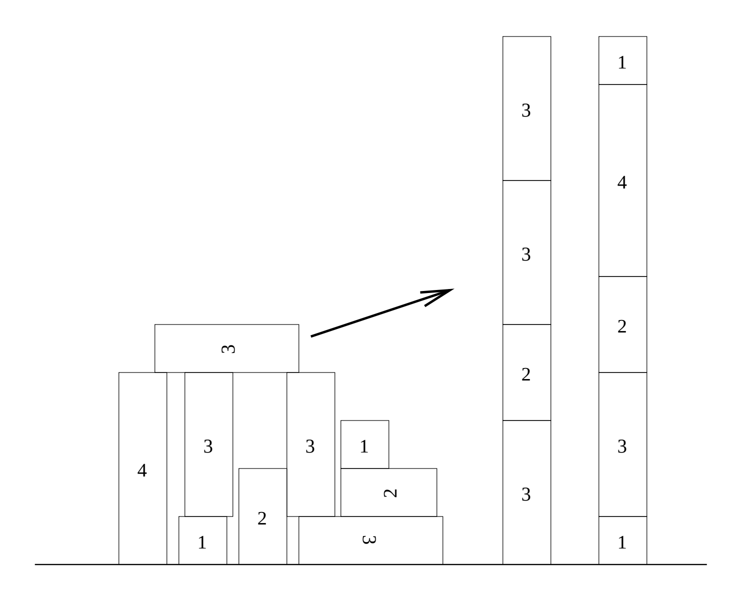


Figura 5.1: Un ejemplo del problema de la repartición

Existe un algoritmo, sin embargo, el cual a primera vista parece resolver el problema de la repartición en forma rápida. Se trata de llenar una tabla de verdad: la entrada (i, j)-ésima es igual a 1 (verdadero) si hay un subconjunto $\{x_1, x_2, \ldots, x_i\}$ cuya suma es j. El algoritmo revisa la tabla, renglón por renglón. Para calcular el valor en la (i, j)-ésima posición, se examinan los valores en las posiciones $(i-1, j)$ e $(i-1, j-x_i)$. Obviamente, si la posición $(i-1, j)$ en la tabla es 1, entonces el subconjunto $\{x_1, x_2, \ldots, x_{i-1}\}$ cuya suma es j debe ser un subconjunto de $\{x_1, x_2, \ldots, x_i\}$. Si, sin embargo, hay un subconjunto de $\{x_1, x_2, \ldots, x_{i-1}\}$ el cual suma $j - x_i$, entonces el mismo subconjunto sumado con x_i debe sumar j.

Una tabla para los bloques de la figura 5.1 se muestra parcialmente a continuación, considerando tomar los bloques en el orden 3, 4, 3, 2, 3, 2, 3, 2, 1. Estos números suman un total de 22, de tal modo que se busca un subconjunto que sume 11.

i j	1	2	3	4	5	6	7	8	9	10	11
x_1	0	0	1	0	0	0	0	0	0	0	0
x_2	0	0	1	1	0	0	1	0	0	0	0
x_3	0	0	1	1	0	0	1	0	0	1	0
x_4	1	0	1	1	1	0	1	1	0	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

El primer renglón de la tabla contiene un solo 1, ya que el único subconjunto no vacío de $\{x_1\}$ es el mismo conjunto, donde $x_1 = 3$. El siguiente renglón contiene tres 1s correspondientes a los subconjuntos $\{x_1\}$, $\{x_2\}$ y $\{x_1, x_2\}$. Estos suman 3, 4, y 7 respectivamente. Para cuando se alcanza el cuarto renglón, se descubre que $x_1 + x_2 + x_3 + x_4 = 11$, que es el valor que se busca originalmente.

De la misma manera en que se describe esta tabla, el siguiente algoritmo va generándola, renglón por renglón. Sin embargo, se presenta una verisón del algoritmo en el cual las posiciones en la tabla no se llenan con 0 ó 1, sino con 0 ó X, donde X es un subconjunto de $\{x_1, x_2, \ldots, x_n\}$ que puede ser diferente de una posición en la tabla a la siguiente. De hecho, X en la posición ij-ésima es un subconjunto (si existe) $\{x_1, x_2, \ldots, x_i\}$ cuya suma es j.

REPARTICION

1. Inicializa todas las posiciones en la tabla a 0
2. **for** $j \leftarrow 1$ **to** $sum/2$
   a) **if** $j = x_1$ **then** $tabla(1, j) \leftarrow \{x_1\}$
3. **for** $i \leftarrow 2$ **to** n
   a) **for** $j \leftarrow 1$ **to** $sum/2$
      1) **if** $tabla(i-1, j) \neq 0$ **then** $tabla(i, j) \leftarrow tabla(i-1, j)$
      2) **if** $x_i \leq j$ **and** $tabla(i-1, j-x_i) \neq 0$
         then $tabla(i, j) \leftarrow tabla(i-1, j-x_i) \bigcup \{x_1\}$
4. **if** $tabla(n, sum/2) \neq 0$
   then output $tabla(n, sum/2)$
   else output "No hay solución"

No parece ser difícil comprobar que este algoritmo es exacto: *Siempre* encuentra una solución (o reporta que "No hay solución") para cualquier partición que se le solicite. Pero, ¿cuánto tarda en hacer esto?

Contando cada **if ... then**, asignación, y salida **output** como 1 unidad de tiempo, se llega al siguiente cálculo:

Línea	**Unidades de tiempo**
1	$n \times (sum/2 + 1)$
2	$sum/2 + 1$
3	n
4	$2(n-1) \times (sum/2 + 1)$
5	1
Total	$3n(sum/2) + 4n - sum/2 + 1$

Para calcular la complejidad de este algoritmo, se eliminan constantes y términos aditivos dominados por el primer término. Por tanto, se dice que el algoritmo *REPARTICION* tiene una complejidad de $O(n\,sum)$. A primera vista, parece como si este algoritmo fuera polinomial respecto al tamaño de su entrada, por lo que resulta útil preguntarse en este punto, ¿de qué tamaño es la entrada del algoritmo?

Originalmente, se dan n enteros $x_1, x_2, \ldots, x_n$. Si se es descuidado, es posible decir que cada entero x_i tiene tamaño x_i, por lo que se puede concluir que la longitud total del conjunto es sum. Ya que $n < sum$, entonces el algoritmo se ejecuta en el tiempo acotado por $O(sum^2)$, lo cual es ciertamente polinomial debido a la medida de la longitud.

De hecho, un requerimiento del problema es que las representaciones sean "concisas", lo que desecha la posibilidad de medir la longitud de esta forma: en general, se requiere que un entero se represente por un esquema cuya longitud sea al menos una función polinomial cuya representación tenga una mínima longitud. Esto resulta ser falso para la medida hecha anteriormente. De hecho, el tamaño s de los enteros $x_1, x_2, \ldots, x_n$ cuando se les escribe en forma binaria sería de:

$$s = log_2 x_1 + log_2 x_2 + \cdots + log_2 x_n$$

Ahora bien, si $n\,sum$ fuera acotado por una función polinomial de s, se podría escribir:

$$n\,sum < s^k$$

para algún número fijo k y para todos los valores de s mayores que otro valor fijo m. Si x_j resulta ser el miembro más grande del conjunto $\{x_1, x_2, \ldots, x_n\}$, entonces:

$$s^k \leq (n\,log_2\,x_j)^k$$

y

$$x_j \leq n\,sum$$

Se sigue a partir de la primera desigualdad que:

$$x_j < n^k (log_2\,x_j)^k$$

para toda $n \geq m$.

Ya que x_j y n son valores independientes, es posible escoger el valor de x_j tan grande como se desee en relación con n, hasta el punto en que se force que n sea mayor que m. De cualquier modo, hasta ahora es todavía fácil seleccionar x_j lo suficientemente grande para que viole la última desigualdad. Por lo tanto, $n\,sum$ no está acotada por ninguna función polinomial de s, y ciertamente el algoritmo no es polinomial en el tiempo.

En contraste con esta conclusión, es posible preguntarse si la cantidad *sum* debería dársele alguna clase de cota predefinida, acordando restringirse a sólo aquellos valores que satisfacen tal cota. Por ejemplo, si se requiere que:

$$sum < s^2$$

entonces se halla un gran número de problemas de interés "práctico". Más precisamente, hay algoritmos pseudoveloces para problemas de planificación de acciones en los cuales los valores a tratar no resultan muy grandes (no tiene caso planificar acciones cuya ejecución requiera un tiempo casi infinito).

En cualquier caso, y con valores que satisfacen la desigualdad, se tiene un algoritmo que se ejecuta en tiempos no peores que $O(s^3)$, ya que:

$$n\, sum < n\, s^2 < s^3$$

Previamente se ha utilizado el término *pseudoveloz* en lugar del término estándar *tiempo pseudopolinomial*. Se dice que un algoritmo se ejecuta en tiempo pseudopolinomial si su complejidad en cada intancia I está superiormente acotada por un polinomio tanto en longitud de I, como en $max\, I$, que se refiere tan solo al tamaño del número más grande en I.

El algoritmo *REPARTICION* es de tiempo pseudopolinomial para resolver el problema de la partición, ya que:

$$\begin{aligned} n\, sum &< n(n\, x) \\ &< n^2\, x \\ &< s^2\, x \end{aligned}$$

donde $x = max\{x_1, x_2, \ldots, x_n\}$ y s es la función de longitud.

Finalmente, se hace mención a otro problema más general que el problema de la repartición. El algoritmo para encontrar una repartición también se utiliza para resolver el problema de la "suma del subconjunto" en un tiempo pseudopolinomial: dados los enteros $x_1, x_2, \ldots, x_m$ y otros entero B, se debe encontrar un subconjunto de m enteros que sume el valor de B. Este problema tiene relevancia en los sistemas de encripción de llaves públicas.

Capítulo 6

Montículos y Mezclas
Los Ordenamientos Más Rápidos

Sería injusto para el lector mencionar en un principio que ningún algoritmo puede ordenar n números en menos que $n\,log\,n$ pasos, sin mostrar al menos uno o dos algoritmos que pueden hacerlo lo suficientemente rápido. Los ejemplos más dramáticos de tales algoritmos son las técnicas conocidas como "ordenamiento por montículos" (*heapsort*) y "ordenamiento por mezcla" (*mergesort*). Estos algoritmos demuestran la eficiencia que algunas veces resulta de una aproximación del tipo divide-y-conquista a un problema dado.

El ordenamiento por montículo depende del concepto de "montículo" (*heap*) que no es mas que una estructura especial de datos. Un ejemplo natural de un montículo podría ser el organigrama de una corporación, en el cual cada empleado tiene una posición numerable dentro de la organización. Lejos de parecer un montículo en el sentido ordinario de la palabra, tal estructura parece más bien un árbol (figura 6.1)

En una organización estable, la posición de un empleado se refleja por su número dentro del organigrama: es mayor que aquellos a los que se supervisa, y menor que quien lo supervisa. Tal árbol es en realidad un montículo.

El algoritmo de ordenamiento por montículo depende de la habilidad de convertir un árbol de números en un montículo. Más aún, mientras más rápido se logre esto, más rápido será el algoritmo. Supóngase, entonces, que se presenta la jerarquía de una organización en la que un número de empleados merecen una promoción. Una forma de resolver esto es seleccionar al azar un empleado que tenga un número mayor que su jefe inmediato, y promoverlo, lo que significa que tal empleado se intercambia con su jefe dentro de la jerarquía. Eventualmente, surge un montículo, pero ¿qué tan eventualmente? Para algunos ejemplos que involucran un número dado de empleados, se ha observado que es posible realizar muchas más promociones

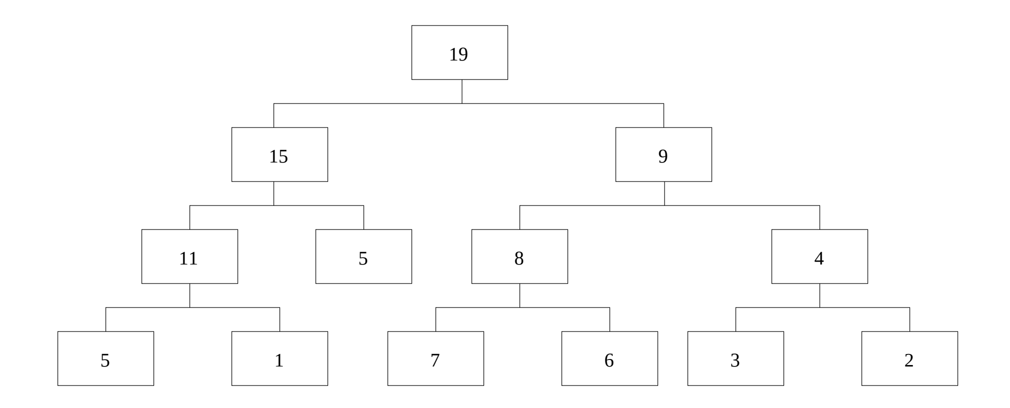


Figura 6.1: Un organigrama

que empleados.

El procedimiento más eficiente para convertir un árbol arbitrario en un montículo se describe simplemente en términos algorítmicos como sigue (figura 6.2):

for each *nodo* X

1. $W \leftarrow max(Y, Z)$
2. **if** $X < W$ **then** intercambia X con el mayor de entre Y y Z

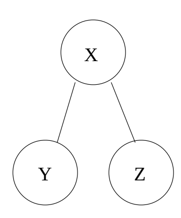


Figura 6.2: Convirtiendo un árbol en montículo

El algoritmo no está del todo completo: no se indica en qué orden debe procesarse los n nodos del árbol. Sin embargo, el orden es crucial. Si el algoritmo trabaja de abajo hacia arriba, únicamente se realizan $2n$ promociones en el peor de los casos. Nótese que en la parte baja del árbol se encuentra un número de pequeños sub-árboles de tres nodos cada uno. Se aplica el procedimiento de promoción a cada uno de éstos. En el siguiente nivel hacia arriba, los sub-árboles tienen siete nodos cada uno. Se aplica entonces el procedimiento de promoción a los tres nodos más altos de estos sub-árboles. En general, la promoción se va aplicando de la forma como se muestra en la figura 6.3

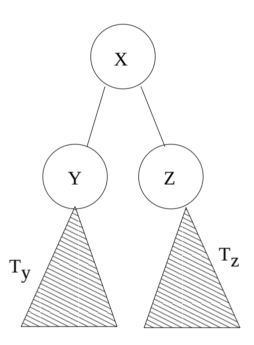


Figura 6.3: ¿Serán T_y y T_z montículos después de un intercambio de X?

Si X es ya mayor que Y y Z, no hay promoción. Ya que los sub-árboles T_y y T_z son montículos, también lo es el sub-árbol X. Si se intercambia X con Y, entonces T_z continúa siendo un montículo. Pero el nuevo sub-árbol T_x (que se obtiene de reemplazar Y por X) puede no ser un montículo: X podría ser menor que alguno de sus descendientes. En tal caso, el algoritmo básico se invoca de nuevo para X, y se continua hasta que X encuentra una posición final.

Parecería que el algoritmo descrito visita varios nodos del árbol más de una vez. ¿Cómo se sabe que no se requieren del orden de n^2 cambios para crear finalmente un montículo? Hay una comprobación simple de que a lo más n promociones son suficientes. Un diagrama hace que la idea central de la comprobación quede clara: supóngase que cada vez que ocurre una promoción, el número reemplazado se lleva por una sucesión de promociones hasta la parte baja del árbol. Para contar el número total de promociones visualmente, no hay problema al considerar que cada cadena de promociones sigue un camino diferente a través del árbol (figura 6.4). El hecho de que tal arreglo sea siempre posible significa que el número total de promociones no es mayor que el número de aristas del árbol. Tal número es, por supuesto, $n - 1$.

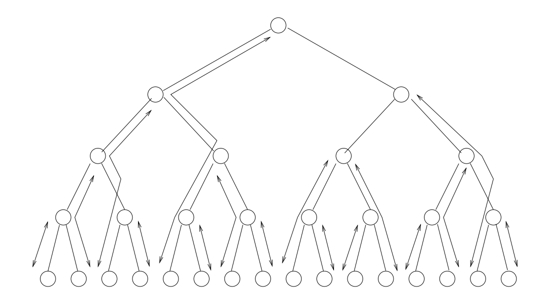


Figura 6.4: Cadenas de promociones en un montículo

Teniendo un procedimiento para convertir un árbol en un montículo, es notorio que el ordenamiento por montículo es directo. Primero, se arreglan los n números arbitrariamente en forma de un árbol binario completo (o al menos casi completo). En seguida, se convierte el árbol en un montículo mediante el algoritmo descrito previamente. El ordenamiento se realiza al remover el número en el nodo raíz, reemplazándolo por un número que a su vez ha sido removido de la parte más baja del montículo. Cada vez que esta operación se realiza, el número raíz sale y el número que lo reemplaza se procesa por el esquema de promoción hasta que encuentra su posición apropiada dentro del árbol.

Los números que van resultando por esta versión del ordenamiento por montículo se encuentran en orden decreciente. El orden opuesto resulta si se utiliza la definición opuesta de montículo: cada número es menor que sus descendientes.

Toma $O(n)$ pasos el convertir un árbol inicial a un montículo. A partir de eso, por cada número que se saca por el nodo raíz, son suficiente $O(log\, n)$ promociones para restaurar el montículo. El número de pasos requeridos, por tanto, es $O(n\, log\, n)$.

La noción de divide-y-conquista se considera en la técnica de montículo en el propio algoritmo de promoción: el número en el nodo raíz de cada sub-árbol se intercambia con al menos uno de sus descendientes. En otras palabras,el impacto de cada promoción se confina en cada paso a tan solo la mitad de los nodos descendientes de un nodo en particular.

La técnica de ordenamiento por mezcla (*mergesort*) también depende fuertemente de una estrategia divide-y-conquista. Si se desea ordenar una secuencia A de n números en orden decreciente, supóngase que los números ya han sido ordenados en dos sub-secuencias del mismo tamaño B y C. Si estas dos sub-secuencias se encuentran ya ordenadas de la manera requerida, ¿qué tan rápido se puede ordenar la secuencia original A? La respuesta es tan rápido como tome mezclar las dos secuencias en una sola. Nótese que el algoritmo de mezcla debe tomar en cuenta los tamaños relativos de las secuencias:

1. $j \leftarrow 1$; $k \leftarrow 1$
2. **for** $i = 1$ **to** n
    a) **if** $B(j) > C(k)$
    then $A(i) \leftarrow B(j)$
    $j \leftarrow j + 1$
    else $A(i) \leftarrow C(k)$
    $k \leftarrow k + 1$

El algoritmo llena las n posiciones de el arreglo A mediante examinar los tamaños relativos de los siguientes números a ser colectados de los arreglos B y C. Si

el número en B es mayor, se le selecciona. De otra manera, se selecciona el número en C. Obviamente, el algoritmo requiere $O(n)$ pasos: para ser exactos, $6n + 2$.

Mezclar dos secuencias previamente ordenadas es, sin embargo, todavía una labor lejana a ordenar una secuencia completamente desordenada. ¿Cuál es el objeto de mezclar? Se puede contestar esta pregunta mediante implícitamente invocar el concepto de divide-y-conquista, al realizar otra pregunta: ¿cómo es que las sub-secuencias ordenadas llegaron a estar ordenadas? La respuesta surge de inmediato: como el resultado de mezclar dos sub-sub-secuencias, de la mitad de la longitud de la sub-secuencia.

Este razonamiento se puede continuar hasta el final, resultando solo $\lceil log\, n \rceil$ etapas. En la primera etapa, dos secuencias de longitud $n/2$ se mezclan en pares. Ciertamente, en cada etapa n números participan (cada uno, una vez) en la operación de mezcla. Es claro que el número total de pasos básicos tomados por el algoritmo implícitamente es $O(n\, log\, n)$.

El algoritmo se llama ordenamiento por mezcla por obvias razones. Se presta especialmente para una formulación recursiva:

MEZCLA

$ordena(i, j)$: **if** $i = j$

1. **then** *regresa*
2. **else** $k \leftarrow (i + j)/2$
   $ordena(i, k)$
   $ordena(k + 1, j)$
   $mezcla(A(i, k), A(k + 1, j))$

$ordena(1, n)$

Este algoritmo supone la existencia de una rutina de *mezcla* como la descrita anteriormente. Tal rutina se encarga de mezclar los elementos de A entre la i-ésima y la k-ésima posiciones con aquéllas entre la $(k+1)$-ésima y la j-ésima posiciones. La recursión entra cuando el procedimiento *ordena* se define en términos de sí mismo:

"Para ordenar los números en las posiciones i-ésima y j-ésima, encuéntrese una aproximación razonable k del índice a la mitad entre i y j. Entonces ordene los números de las posiciones i-ésima a la k-ésima, y de la $(k + 1)$-ésima a la j-ésima, y mezcle las dos secuencias (o arreglos) que se han formado."

La instrucción final es una llamada simple, $ordena(1, n)$, que invoca al procedimiento dando como datos $i = 1$ y $j = n$. Esto dispara una cascada de llamadas dentro de este procedimiento cada vez que se auto-invoca, dos veces por llamada. En términos esquemáticos, por ejemplo, la secuencia 2, 8, 5, 3, 9, 1, 6 se ordena como se muestra en la figura 6.5.

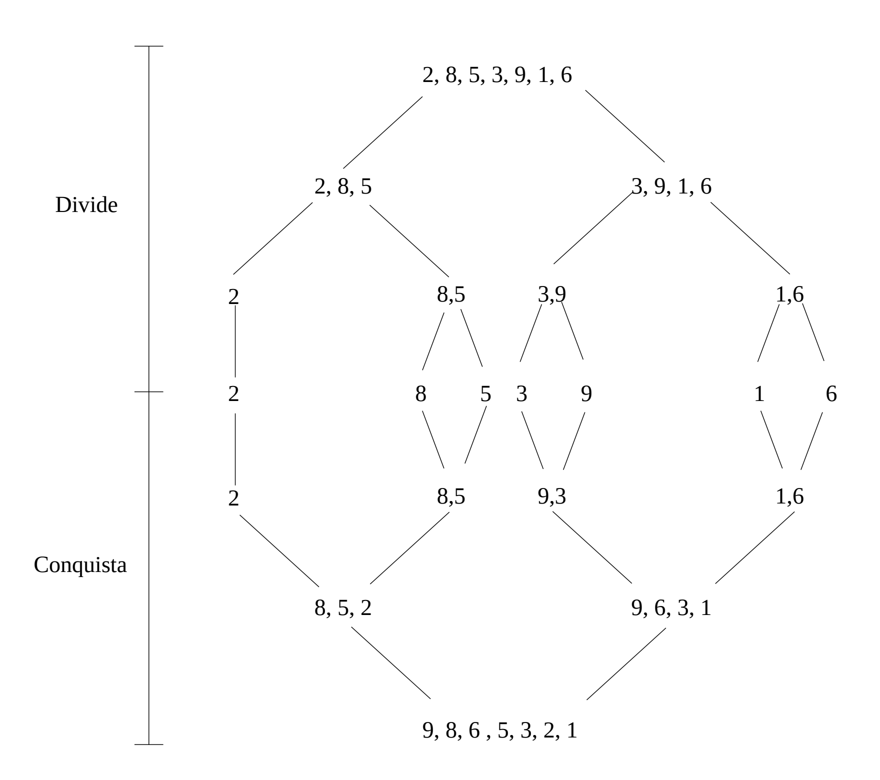


Figura 6.5: Ordenamiento por mezcla

En la fase de "divide" del ordenamiento por mezcla, el arreglo de entrada A se parte y sub-parte hasta dejar elementos individuales del arreglo. En la fase de "conquista", el procedimiento comienza una larga secuencia de retornos a las llamadas anteriormente realizadas a sí mismo. En otras palabras, los ordenamientos más internos se han completado, y los elementos comienzan un proceso de mezcla, al principio de dos en dos, luego de cuatro en cuatro, y así hasta terminar con todos los elementos del arreglo.

Ni la estructura de montículo ni la operación de mezcla parecen a primera vista ser elementos clave para un algoritmo de ordenamiento, y sin embargo, lo son. En ambos casos, la aproximación divide-y-conquista lleva a un factor logarítmico para el tiempo de ordenamiento. Esto simplemente no puede ser mejorado utilizando computadoras secuenciales.

Capítulo 7

Detectando Primos

Un Algoritmo que Casi Siempre Funciona

Nadie hasta ahora ha desarrollado un algoritmo para decidir en tiempo polinomial si un número es primo. ¿Existe un polinomio p y un algoritmo A tales que, para cualquier entero positivo n, A pueda descubrir si n es primo en tan solo $p(\lceil log\, n \rceil)$ pasos? Aquí, se utiliza $\lceil log\, n \rceil$ como una medida de la longitud de la entrada, ya que se supone que n se representa en notación binaria.

Como muchos problemas abiertos, tan pronto como el tamaño de la entrada se hace moderadamente grande, los algoritmos conocidos en la actualidad para resolverlos simplemente requieren de mucho tiempo. Por ejemplo, nadie tiene la paciencia de esperar mientras un algoritmo trata de decidir si un número en el vecindario de 2^{400} es primo. Parecería que para algunos problemas, al menos, este es el precio que hay que pagar por una certeza absoluta en la respuesta, siempre y cuando ésta llegue.

Imagínese, entonces, un algoritmo que, en el lapso de *unos cuantos minutos*, decide que $2^{400} - 593$ es primo con la probabilidad de:

0.9999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999

Un algoritmo como este, especialmente si es fácil de entender y de programar, tendría un atractivo tanto práctico como estético.

De hecho, tal algoritmo sí existe. Fue descubierto por Michael O. Rabin, y depende de la noción de que los enteros sean “testigos” de la “composabilidad” de un número n. Si un solo testigo puede hallarse, n queda como un número compuesto; pero si durante un tiempo razonable se busca por un testigo y no se encuentra ninguno, entonces se dice que n tiene el estatus de “primalidad”, y efectivamente lo mantiene mientras no se encuentre un testigo.

Rabin define un *testigo de la composabilidad de n* a cualquier entero w que satisface las siguientes condiciones:

- $w^{n-1} = 1 \, mod \, n$
- Para algún entero k,

$$1 < mcd(w^{(n-1)/2^k}) - 1, n) < n$$

Resulta fácil notar que la existencia de un testigo significa que n es compuesto porque, en la segunda condición, el máximo común divisor (mcd) de n (y de cualquier otro número) es ciertamente un factor de n. Es también fácil de desarrollar un algoritmo que cheque en tiempo polinomial si un entero dado es un testigo de la composabilidad de n.

Ya que puede determinarse rápidamente si un número es un testigo, solo queda preguntar qué tan comunes son los testigos. Ciertamente, si los testigos son poco comunes, se podría difícilmente usar nuestra inhabilidad para encontrar un testigo como una base para considerar que n es primo. Es aquí donde un teorema de Rabin se hace muy útil:

> *Teorema:* Si n es un número compuesto, entonces más de la mitad de los números en el conjunto $\{2, 3, \ldots, n-1\}$ son testigos de la composabilidad de n.

Resulta fácil ahora esquematizar un algoritmo para probar si un número n es primo:

PRIMO

1. **Input** n
2. Selecciona m enteros $w_1, w_2, \ldots, w_m$ al azar del conjunto $\{2, 3, \ldots, n-1\}$
3. **for** $i \leftarrow 1$ **to** m
   probar si w_i es un testigo
4. **if** m es testigo
   then output “SI”
   else output “NO”

Ya que más de la mitad de los enteros en el conjunto $\{2, 3, \ldots, n-1\}$ son testigos de la composabilidad de n, en el caso de que n no sea primo la probabilidad de que ningún testigo sea seleccionado al azar es de $(1/2)^m$. En otras palabras, el algoritmo tiene tan solo una pequeña posibilidad de fallar en la detección de la composabilidad de n, específicamente si se escoge un valor grande para m. Por lo tanto, si el algoritmo arroja un "SI", la probabilidad *a priori* de que n sea primo es claramente de al menos $1 - (1/2)^m$.

En el ejemplo utilizado al inicio de esta nota, se utiliza un valor de $m = 400$. Cuando el número de testigos seleccionados para una prueba tiene cerca del mismo orden de magnitud del tamaño del problema, por ejemplo $\lceil log\, n \rceil$, el algoritmo puede arrojar un "SI" o un "NO" en un tiempo razonablemente corto. Un experimento interesante llevado a cabo por Rabin involucró una prueba de todos los números de la forma $2^p - 1$, para $p = 1, 2, \ldots, 500$, con solo 10 testigos en cada caso. La probabilidad de error fue de apenas menos que $1/1000$, y en el lapso de algunos minutos, el algoritmo fue ejecutado; los "SI" arrojados como resultado coincidieron *exactamente* con la tabla de primos de Mersenne, que son primos de la forma $2^p - 1$.

Por supuesto, siempre hay un sentimiento de intranquilidad y duda tras haber ejecutado el algoritmo de Rabin para un número n, y descubrir que tal número es primo: ¿realmente lo es? Se estaría tentado a incrementar k al punto en que el tiempo que se requiere para confirmar que n es primo llega a varias horas en lugar de minutos. Se propone entonces una regla para terminar la ejecución: valores bastante moderados de k son suficientes para garantizar que el *hardware* de la computadora tiene mayor probabilidad de fallar antes de que falle el propio algoritmo.

Tan interesante y atractivo como pudiera parecer la aproximación de Rabin para problemas abiertos, es necesario hacer una advertencia: uno puede fácilmente inventar testigos para la mayoría de las clases de problemas. Sin embargo, pueden ser poco comunes, y aún si son relativamente comunes, resulta muy difícil probar que lo sean.

Capítulo 8

Computación en Paralelo
Procesadores con Conexiones

Imagínese n computadoras simples (llamadas comúnmente *nodos* o *procesadores*), organizados como algún tipo de arreglo de tal manera que cada procesador sea capaz de intercambiar información sólo con sus nodos vecinos. El caso más simple resulta ser el conectarlos en línea, donde cada procesador sólo tiene dos vecinos: uno a la derecha y otro a la izquierda; los procesadores a los extremos, pueden servir como elementos de entrada y salida. Tal tipo de "computadora" constituye el ejemplo más simple de lo que se conoce como una *máquina sistólica* (figura 8.1). En general, puede resolver varios problemas de forma más rápida que una máquina con un solo procesador. Uno de estos problemas es el famoso "problema de los n cuerpos" (*n-body simulation*): calcular el recorrido de cada uno de n cuerpos que se mueven a través del espacio bajo la influencia de sus atracciones gravitacionales mutuas y combinadas.

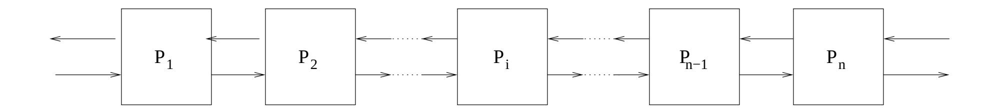


Figura 8.1: Un arreglo sistólico

Una computadora secuencial (con un solo procesador) puede llevar a cabo el cálculo de todas las $(n^2 - n)/2$ atracciones en $O(n^2)$ pasos básicos. Una máquina sistólica, sin embargo, puede lograr el mismo resultado en $O(n)$ pasos básicos, lo que significa una mejora en velocidad en un factor de n. Funciona como se describe a continuación.

El ciclo básico del cómputo del problema de los n cuerpos involucra calcular para cada uno de los cuerpos la suma de las atracciones de los otros $n-1$ cuerpos. Es razonable suponer, para una máquina sistólica, que cada procesador cuenta con un programa para calcular la fórmula de Newton:

$$F_{ij} = k\, m_i m_j / {d_{ij}}^2$$

donde m_i y m_j son las masas del i-ésimo y j-ésimo cuerpos, respectivamente, k es la constante gravitacional, y d_{ij} es la distancia entre ellos. A cada etapa del cómputo, cada procesador calcula la distancia entre dos cuerpos mediante la fórmula euclideana estándar dadas las coordenadas de los dos cuerpos, y calcula la fuerza entre ellos mediante la fórmula anterior.

El cómputo comienza cuando las coordenadas y la masa del n-ésimo cuerpo B_n se introducen al procesador P_1. En seguida, tales datos se pasan al segundo procesador P_2, en tanto que se introducen las coordenadas y masa del $(n-1)$-ésimo cuerpo B_{n-1} al procesador P_1. Esto continúa hasta que cada procesador tiene las coordenadas y la masa de un cuerpo único del problema gravitacional. En general, se puede decir que P_i mantiene las coordenadas y masa del cuerpo B_i. Obviamente, este proceso absorbe n ciclos de transmisiones.

La segunda fase del cálculo en una máquina sistólica involucra exactamente la misma secuencia de entrada, solo que ahora cada procesador ya tiene "cargada" la información de su cuerpo asignado. Conforme la información de cada nuevo cuerpo B_j llega por la izquierda, el procesador P_i ejecuta el siguiente algoritmo:

1. Calcular d_{ij}.
2. Calcular la fuerza F_{ij}.
3. Sumar F_{ij} a la fuerza previamente calculada.

Esto simplifica en mucho el modelo, considerando tan solo una suma de componentes de fuerza por cada cuerpo. Sin embargo, es claro y notorio también que solo una cantidad constante de tiempo se absorbe antes de que cada procesador esté listo para la siguiente ronda de datos.

Después de $2n$ pasos, cada procesador ha considerado $n-1$ fuerzas, y una nueva parte del programa hace recorrer los valores obtenidos para su salida en el otro extremo del arreglo de procesadores. Este recorrido absorbe otros n pasos.

Si se cuentan como que cada corrimiento de información absorbe 1 unidad de tiempo, y se cuenta que la ejecución del algoritmo descrito toma también 1 unidad de tiempo, entonces todo el proceso de cómputo toma en total solo $4n$ pasos, que representa una gran mejora respecto a una computadora secuencial.

Las máquinas sistólicas pueden ser mucho más sofisticadas que el arreglo de procesadores utilizado para el problema de los n cuerpos. Por ejemplo, una geometría de procesadores comúnmente utilizada y discutida forma una malla cuadrada

o rectangular. De forma similar, en el contexto del procesamiento paralelo, las máquinas sistólicas son apenas una clase dentro de un vasto rango de tipos de computadoras paralelas que se han construido o que se encuentran en desarrollo. Un ejemplo representativo de esquemas más generales y poderosos es la computadora hipercúbica.

Un hipercubo d-dimensional forma la base para las conexiones entre n procesadores. Cada procesador ocupa un vértice del cubo o hipercubo. El número total de n procesadores es por lo tanto 2^d. Por supuesto, este arreglo se refiere únicamente a la topología de conexiones entre los procesadores, y no a la geometría real. Para enfatizar este punto (que por cierto, se aplica de la misma forma a las máquinas sistólicas) es posible arreglar todos los n procesadores en un cuadrado bidimensional, como se muestra en la figura 8.2.

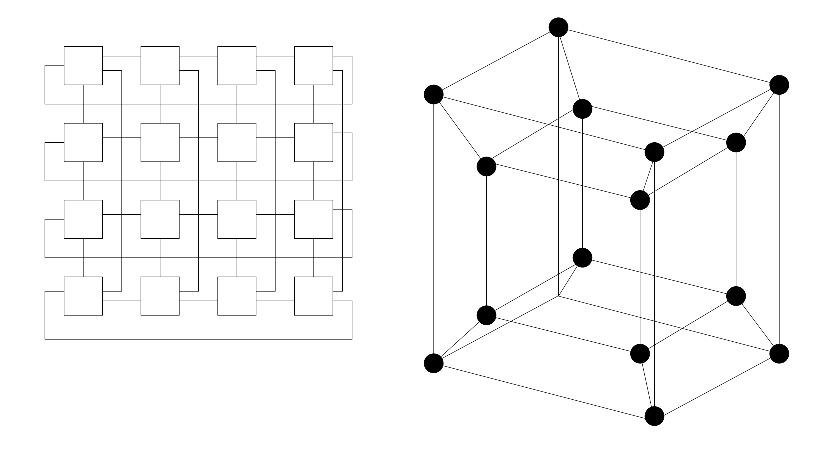


Figura 8.2: Dos formas para un hipercubo

A continuación, se discute la solución de un problema práctico que se ha mencionado anteriomente, pero utilizando una computadora conectada en hipercubo: la multiplicación entre matrices. Dadas dos matrices X y Y de $n \times n$, ¿qué tan rápido se pueden formar los n^2 elementos de la matriz producto Z? En el capítulo 4 se llega a la conclusión de que tal producto puede realizarse entre $O(n^2)$ y $O(n^3)$. Para la solución paralela, el tiempo se reduce a $O(log\, n)$ pasos.

Hasta cierto punto, es conveniente añadir procesadores para ciertos problemas. Para llevar a cabo la multiplicación entre matrices, se requiere de n^3 procesadores. De nuevo, no hay ningún problema en considerar que n^3 es una potencia de 2. En cualquier caso, antes de revisar el algoritmo de multiplicación de matrices en paralelo, vale la pena observar que se puede requerir hasta d conexiones separadas e independientes, de tal modo que los procesadores puedan comunicarse entre sí y en forma simultánea.

Para el caso de n^3 procesadores conviviendo en un cubo de dimensión d, se debe cumplir que:

$$d = log\, n^3 = 3\, log\, n$$

En otras palabras, la mera comunicación entre procesadores requiere de $O(log\, n)$ pasos en el tiempo.

Ahora bien, se sabe que el ij-ésimo elemento de la matriz producto Z tiene la forma:

$$z_{ij} = \sum_{k=1}^{n} x_i y_{kj}$$

Por coincidencia, el cómputo paralelo de este ejemplo procede en tres fases, al igual que el ejemplo de la máquina sistólica. La primer fase distribuye los elementos de los arreglos X y Y entre los n^3 procesadores. La segunda fase realiza los productos. Finalmente, la tercera fase lleva a cabo las sumatorias.

Es conveniente identificar a cada procesador mediante un índice de tres números (k, i, j). Cada índice puede tener n valores binarios consecutivos. De este modo, cada procesador $P(k, i, j)$ se conecta únicamente a los procesadores que difieran exactamente en un bit de sus valores de índice. Por ejemplo, el procesador $P(101, i, j)$ debe estar conectado con el procesador $P(001, i, j)$.

Inicialmente, los elementos x_{ij} y y_{ij} se depositan en el procesador $P(0, i, j)$. La primera fase, entonces, se encarga de distribuir estos datos por los procesadores del hipercubo, de modo que en general el procesador $P(k, i, j)$ contiene a x_{ik} y a y_{kj}. El procedimiento descrito aquí solo considera el caso de x_{ik}, pero el caso de y_{kj} es similar.

Primero, el contenido x_{ik} de $P(0, i, k)$ se transmite a $P(k, i, k)$ por una ruta a través de otros procesadores menor a $log\, n$. El mensaje mismo consiste en el valor x_{ik} y el índice del procesador objetivo (k, i, k). A cada paso del viaje del mensaje, el siguiente procesador al que visita es aquél que tiene un índice con 1 bit más cercano a k. Por ejemplo, si $k = 5 = 101$, entonces la secuencia que se sigue podría ser:

$$P(000, i, k) \rightarrow P(100, i, k) \rightarrow P(101, i, k)$$

En seguida, cuando el procesador $P(0, i, k)$ ha enviado su mensaje a un procesador $P(k, i, k)$ (en paralelo), éste re-envía el mismo mensaje a todos los demás procesadores $P(k, i, 1), P(k, i, 2), \ldots, P(k, i, n)$. Esto se logra mediante un tipo de emisión (*broadcasting*). El mensaje se envía simultáneamente a través de las conexiones de comunicación en los cuales uno de los bits relevantes difiere del bit actual. Por ejemplo, si $k = 101$, como se dijo anteriormente, el mensaje podría ser enviado dependiendo del patrón paralelo que se muestra en la figura 8.3. Aquí, de nuevo, el tiempo requerido para la transmisión es menor que $O(log\, n)$ pasos.

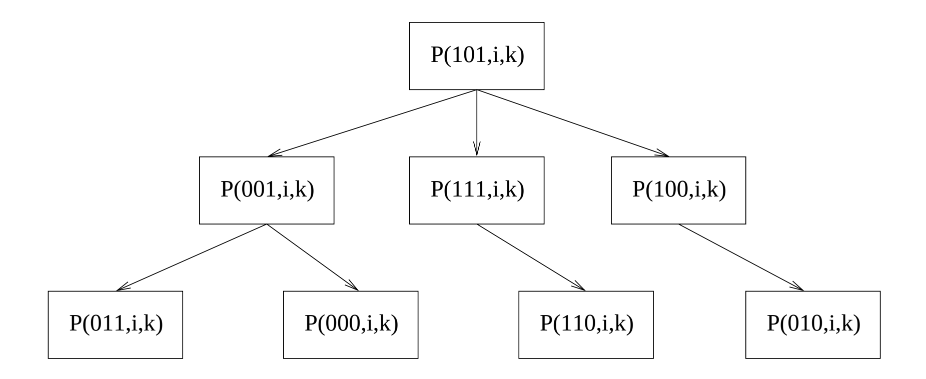


Figura 8.3: Distribuyendo elementos de las matrices

Con los elementos de las matrices x_{ik} y y_{kj} en cada procesador $P(k, i, j)$, la segunda fase del cómputo paralelo toma lugar: el producto $x_{ik} \times y_{kj}$ se realiza y se almacena en el mismo procesador.

La tercera fase, como la primera, es algo complicada. Esencialmente, los productos son llevados de todos los procesadores $P(1, i, j), P(2, i, j), \ldots, P(n, i, j)$ al procesador $P(0, i, j)$ por un "sumidero" de sumas acumuladas. De nuevo, un mensaje se transmite de un procesador a otro, pero siempre en la dirección en que se cambia 1 bit del primer índice original al índice 0. Cuando dos de tales sumas llegan al mismo procesador, se suman al producto local y se retransmiten. Supóngase, por ejemplo, que $n = 4$ y que los productos 8, 7, 5, 3, 9, 12, y 6 han sido apenas calculados en $P(1, i, j)$ a través de $P(7, i, j)$, respectivamente. La figura 8.4 muestra un conjunto de las rutas posibles de la suma hasta alcanzar $P(0, i, j)$. En esta fase del cómputo, toda transmisión y suma se realiza en paralelo, con el número de pasos acotado meramente por la distancia máxima entre dos procesadores, es decir, $log\, n$. De esta forma, el producto de dos matrices de $n \times n$ se completa en tan solo $O(log\, n)$ pasos.

La prospectiva del procesamiento paralelo ha generado nuevos desarrollos en el campo del análisis de algoritmos. Los algoritmos paralelos existen ahora para casi cualquier problema clásico de programación. Un marco razonable en el cual estudiar algoritmos paralelos involucra determinar para cada problema si pertenece a un conjunto llamado "clase de Nick" (*Nick's class*[1]). Para calificar como miembro, un problema debe poder resolverse en tiempo *polylog* (el polinomio de un logaritmo) por un número polinomial de procesadores.

[1] "Nick" se refiere a Nicholas Pippenger, un científico de la computación de los laboratorios de IBM en San José, California.

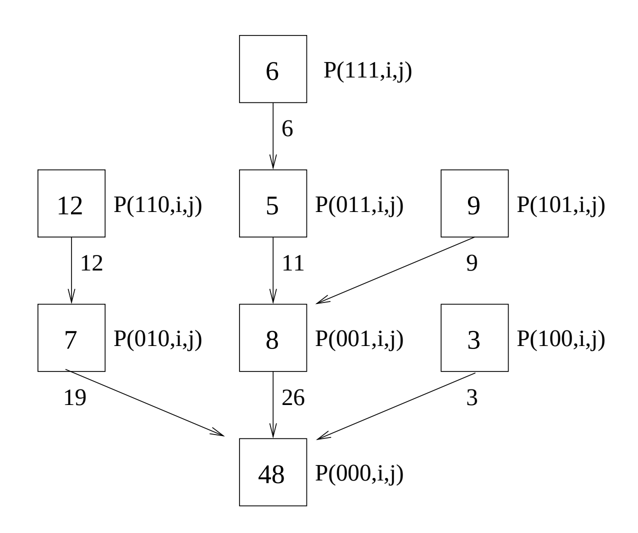


Figura 8.4: Sumando los resultados

Específicamente, esto implica las siguientes reglas: dado un cierto problema P, debe haber dos polinimios p y q tales que para una instancia x de P con tamaño n, existe un algoritmo que, al ejecutarse en $p(n)$ procesadores, resuelve x en tiempo $q(log\, n)$. Se ha presentado aquí un ejemplo de tales problemas: la multiplicación de matrices realizada en n^3 procesadores, que requiere de $O(log\, n)$ pasos para resolverse. En este caso, p es un polinomio cúbico, y q es un polinomio lineal.

Bibliografía

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] R.B. Anderson. *Proving Programs Correct.* Wiley, 1979.

[3] N. Christofides. *Graph Theory: An Algorithmic Approach.* Academic Press, 1975.

[4] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[5] D. Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley, 1987.

[6] K. Hwang. *Supercomputers: Design and Applications.* IEEE Computer Society Press, 1984.

[7] D.E. Knuth. *The Art of Computer Programming, vol. II, Seminumerical Algorithms.* Addison-Wesley, 1967.

[8] D.E. Knuth. *The Art of Computer Programming, vol. III, Sorting and Searching.* Addison-Wesley, 1967.

[9] G.J. Lipovski and M. Malek. *Parallel Computing: Theory and Comparisons.* Wiley, 1987.

[10] M.O. Rabin. *Probabilistic Algorithms.* Algorithms and Complexity, New Directions and Recent Trends (J.F. Taub, editor) Academic, 1976.

[11] E.M.Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

[12] J.E. Savage. *The Complexity of Computing.* Wiley-Interscience, 1976.

[13] T.A. Standish. *Data Structure Techniques.* Addison-Wesley, 1980.